# 4 Building a simple OAuth protected resource

***This chapter covers***

- Parsing an incoming HTTP request for OAuth tokens
- Responding to token errors
- Serving requests differently based on scopes
- Serving requests differently based on the resource owner

Now that we've got a working OAuth client, it's time to create a protected resource for the client to call with those access tokens. In this chapter, we'll be building a simple resource server that our client can call and our authorization server can protect. We'll be giving you a fully functional client and authorization server for each exercise, all designed to work together.

> **NOTE**: All of the exercises and examples in this book are built using Node.js and JavaScript. Each exercise consists of several components designed to run on a single system accessible from *localhost* on various ports. For more information about the framework and its structure, see appendix A.

For most web-based APIs, adding an OAuth security layer is a lightweight process. All that the resource server needs to do is parse the OAuth token out of the incoming HTTP request, validate that token, and determine what kinds of requests the token is good for. Because you're reading this chapter, chances are that you've got something already built or designed that you want to protect with OAuth. For the

exercises in this chapter, we're not expecting you to develop an API on your own just to practice protecting it; instead, we've supplied a handful of resource endpoints and data objects for you to use, and the client application included with each exercise is set up to be able to call these. In this exercise, our resource server will be a simple data store that serves JSON objects to HTTP GET and POST requests on several different URLs, depending on the exercise.

Although the protected resource and authorization server are conceptually separate components in the OAuth structure, many OAuth implementations co-locate the resource server with the authorization server. This approach works well when there's a tight coupling between the two systems. For the exercises in this chapter, we'll be running the protected resource in a separate process on the same machine, but giving it access to the same database used by the authorization server. We'll look at ways to pull this connection apart a little bit more in chapter 11.

## 4.1 Parsing the OAuth token from the HTTP request

Open up the exercise `ch-4-ex-1` and edit the file `protectedResource.js`. You can leave the `client.js` and `authorizationServer.js` files untouched for this exercise.

Our protected resource expects OAuth bearer tokens, which makes sense because our authorization server generates bearer tokens. The OAuth Bearer Token Usage specification[1] defines three different methods for passing bearer tokens to the protected resource: the HTTP Authorization header, inside a form-encoded POST body, and as a query parameter. We're going to set up our protected resource to take any of these three, with a preference for the authorization header.

Since we'll be doing this on multiple resource URLs, we're going to be scanning for the token in a helper function. The Express.js web application framework that our exercises are built on gives us a fairly simple means for doing this, and although the details of the implementation here are particular to Express.js, the general concepts should be applicable to other web frameworks. Unlike most of our HTTP handling functions so far, our helper function is going to take in three parameters. The third parameter, `next`, is a function that can be called to continue processing the request. This lets us chain together multiple functions to serve a single request, allowing us to add this token scanning functionality into other handlers throughout our application. Right now, the function is empty, and we'll be replacing that code in a moment.

```
var getAccessToken = function(req, res, next) {

});
```

The OAuth Bearer token specification tells us that when the token is passed as an HTTP Authorization header, the value of the header consists of the keyword Bearer, followed by a single space, and followed by the token value itself. Furthermore, the OAuth specification tells us that the `Bearer` keyword is not case sensitive. Additionally,

[1] RFC 6750 https://tools.ietf.org/html/rfc6750

the HTTP specification tells us that the `Authorization` header keyword is itself not case sensitive. This means that all of the following headers are equivalent:

```
Authorization: Bearer 987tghjkiu6trfghjuytrghj
Authorization: bearer 987tghjkiu6trfghjuytrghj
authorization: BEARER 987tghjkiu6trfghjuytrghj
```

First, we try to get the Authorization header if it's been included in the request, then see whether it contains an OAuth Bearer token. Since Express.js automatically lowercases all incoming HTTP headers, we'll check against the string literal `authorization` on our incoming request object. We'll also check the `bearer` keyword by similarly converting the value of the header to lowercase.

```
var inToken = null;
var auth = req.headers['authorization'];
if (auth && auth.toLowerCase().indexOf('bearer') == 0) {
```

If both of these pass, we now need to retrieve the token value from the header by stripping off the `Bearer` keyword and the space that follows it. Everything else in the header is the value of the OAuth token, with no further processing needed. Thankfully, this string operation is trivial in JavaScript and most other languages. Notice that the *token value itself is case sensitive,* so we slice the original string and not a transformed version.

```
inToken = auth.slice('bearer '.length);
```

Next, we'll handle tokens passed as form-encoded parameters in the body. This method isn't recommended by the OAuth specification because it artificially limits the input of the API to a form-encoded set of values. If the API natively speaks JSON as inputs, this prevents a client application from being able to send the token along with the input. In such cases, the Authorization header is preferred. But for APIs that do take in form-encoded inputs, this method provides a simple and consistent way for clients to send the access token without having to deal with the Authorization header. Our exercise code is set up to automatically parse an incoming form body, so we have to check whether it exists and pull the token value from it in an additional clause of the previous `if` statement.

```
} else if (req.body && req.body.access_token) {
   inToken = req.body.access_token;
```

Finally, we'll handle the token being passed as a query parameter. This method is recommended by OAuth only as a last resort when the other two methods aren't applicable. When this method is used, the access token has a higher probability of being inadvertently logged in server access logs or accidentally leaked through referrer headers, both of which replicate the URL in full. However, there are situations in which a client application can't access the Authorization headers directly (due to platform or library access restrictions) and can't use a form-encoded body parameter (such as with HTTP GET). Additionally, the use of this method allows a URL to include not only the locator for the resource itself but also the means required to access it. In these cases,

with the appropriate security considerations in mind, OAuth does allow the client to send the access token as a query parameter. We handle it in the same manner as the previous form-encoded body parameter.

```
} else if (req.query && req.query.access_token) {
   inToken = req.query.access_token
}
```

With all three methods in place, our function looks like listing 5 in appendix B.

Our incoming access token value is stored in the `inToken` variable, which is `null` if no token was passed in. However, that's not quite enough: we still need to figure out whether the token is valid and what it's good for.

## 4.2 Validating the token against our data store

For our example application, we have access to the database that our authorization server uses to store tokens. This is a common setup for small installations of OAuth, in which the authorization server and the API that it protects are co-located with each other. The specifics of this step are particular to our implementation, but the technique and pattern are generally applicable. We'll be looking at alternatives to this local lookup technique in chapter 11.

Our authorization server uses a NoSQL database stored in a file on disk with a simple Node.js module to allow access. If you'd like to look at the contents of the database live while the program is in operation, monitor the `database.nosql` file in the exercise directory. Note that editing this file by hand is dangerous while the system is running. Luckily, resetting the database is as simple as deleting the `database.nosql` file and restarting the programs. Note that this file isn't created until the authorization server stores a token in it the first time, and its contents are reset every time the authorization server is restarted.

We'll perform a simple lookup in our database to find the access token based on its incoming value. Our server stores each access token and refresh token as a separate element in the database, so we just have to use our database's search capabilities to find the right one. The details of this search function are specific to our NoSQL database, but the same kind of lookup method can be used for other databases as well.

```
nosql.one(function(token) {
   if (token.access_token == inToken) {
        return token;
   }
}, function(err, token) {
   if (token) {
        console.log("We found a matching token: %s", inToken);
   } else {
        console.log('No matching token was found.');
   }
   req.access_token = token;
   next();
   return;
});
```

The first function passed in checks the value of the stored access tokens against the input token that we pulled off the wire. If it finds a match, it returns the token and the searching algorithm stops. The second function is called when either a match is found or the database is exhausted, whichever comes first. If we do find a token in the store, it will be passed in the `token` argument. If we are unable to find a token with the input value, this argument will be `null`. Whatever we find, we attach it to the `access_token` member of the `req` object and call the `next` function. The `req` object is automatically passed to the next part of the process handler.

The token object that comes back is exactly the same object that was inserted by our authorization server when the token was generated. For example, our simple authorization server stores access tokens and their scopes in a JSON object like this one:

```
{
   "access_token": "s9nR4qv7qVadTUssVD5DqA7oRLJ2xonn",
   "clientId": "oauth-client-1",
   "scope": ["foo"]
}
```

**Do I have to share my database?**

Although working with a shared database is a very common OAuth deployment pattern, it's far from the only one available to you. There's a standardized web protocol called Token Introspection that the authorization server can offer, allowing the resource server to check the token's state at runtime. This lets the resource server treat the token itself as opaque, just like the client does, at the expense of more network traffic. Alternatively, or even additionally, the tokens themselves can contain information that the protected resource can parse and understand directly. One such structure is a JSON Web Token, or JWT, which carries a set of claims in a cryptographically protected JSON object. We'll cover both of these techniques in chapter 11.

You may also wonder whether you have to store your tokens as raw values in the database, as our example setup does. Although this is a simple and common approach, there are alternatives. For example, you can store a hash of the token value instead of the value itself, similar to how user passwords are usually stored. When the token needs to be looked up, its value is hashed again and compared against the contents of the database. You could instead add a unique identifier inside your token and sign it with the server's key, storing only the unique identifier in the database. When the token must be looked up, the resource server can validate the signature, parse the token to find the identifier, and look up the identifier in the database to find the token's information.

After adding this in, our helper function looks like listing 6 in appendix B.

Now we need to wire it into our service. With our Express.js application, we have two main options: wire it to every request, or wire it specifically to the requests that we want to check for OAuth tokens. To have this processing done on every request, we set up a new listener and hook up our function. This needs to be connected before any

other functions in our router since they're processed in the order that they're added in the code.

```
app.all('*', getAccessToken);
```

Alternatively, we can insert our new function into an existing handler setup so that our function is called first. For example, in the code we currently have this function.

```
app.post("/resource",  function(req, res){

});
```

All we need to do to have our token processor function called first is to add our function to the route before the handler definition.

```
app.post("/resource", getAccessToken, function(req, res){

});
```

By the time the handler is called, the request object will have an `access_token` member attached to it. If the token was found, this will contain the token object from the database. If the token was not found, this will contain `null`. We can branch our code accordingly.

```
if (req.access_token) {
   res.json(resource);
} else {
   res.status(401).end();
}
```

Now running the client application and telling it to fetch the protected resource should yield a screen like the one in figure 4.1.

Trying to call the protected resource from our client without the access token will yield an error message, propagated from the HTTP response that the client received back from the protected resource (figure 4.2).

Now we have a very simple protected resource that can decide to fulfill requests or not based on the presence or absence of a valid OAuth token. Sometimes that's


**Figure 4.1 The client's page when it successfully accesses the protected resource**


**Figure 4.2 The client's page when it receives an HTTP error from the protected resource**

enough, but OAuth gives you the opportunity to have much more flexibility in applying security to your protected API.

## 4.3 Serving content based on the token

What if your API isn't serving static resources with a simple yes/no gateway in front of them? Many APIs are designed such that different actions on the API require different access rights. Some APIs are designed to give different answers depending on whose authority the call was made, or give a subset of information based on different access rights. We're going to build a few of these setups here, based on OAuth's scope mechanism and references to the resource owner and client.

In each of the exercises that follow, if you look at the code for the protected resource server, you'll see that we've already included the `getAccessToken` utility function from the last exercise, and we've wired it up to all of the HTTP handlers. However, that function only extracts the access token and doesn't make processing decisions based on its absence or presence. To help with that, we've also wired up a simple utility function called `requireAccessToken` that handles returning the error if the token isn't present, but otherwise passes control to the final handler for further processing.

```
var requireAccessToken = function(req, res, next) {
  if (req.access_token) {
      next();
  } else {
      res.status(401).end();
  }
});
```

In each of these exercises, we'll be adding code to check the status of the token for each of the handlers and returning results appropriately. We've wired up the client in each exercise to be able to ask for all of the appropriate scopes, and the authorization server will let you, acting as the resource owner, decide which ones to apply to a given transaction (figure 4.3).

The client in each exercise also has the ability to call all of the protected resources in a given exercise using different buttons. All of the buttons will be available at all times, regardless of the scope of the current access token.

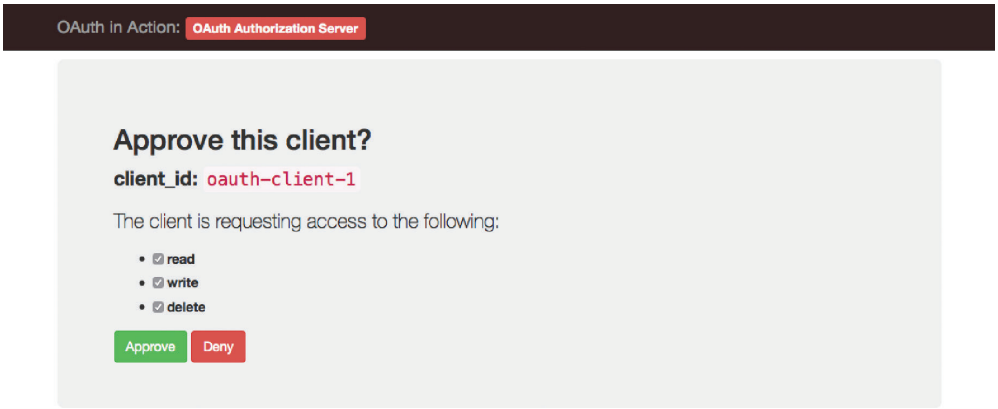


**Figure 4.3 The approval page showing different scopes, available to be checked off**

### 4.3.1 *Different scopes for different actions*

In this style of API design, different kinds of actions require different scopes in order for the call to be successful. This allows the resource server to divide functionality based on what the client is allowed to do. This is also a common way to have a single access token be applicable across multiple resource servers associated with a single authorization server.

Open up `ch-4-ex-2` and edit `protectedResource.js`, leaving `client.js` and `authorizationServer.js` alone. The client has a page that will allow you to access all of the functions in the API, once you get a token (see figure 4.4). The blue button will read the current set of words and display them, along with a timestamp. The orange button will add a new word to the current list stored at the protected resource. The red button will delete the last word in the set.

Three routes are registered in our application, each mapped to a different verb. All of them currently run if a valid access token of any type is passed in.

```
app.get('/words', getAccessToken, requireAccessToken, function(req, res) {
  res.json({words: savedWords.join(' '), timestamp: Date.now()});
});

app.post('/words', getAccessToken, requireAccessToken, function(req, res) {
  if (req.body.word) {
       savedWords.push(req.body.word);
  }
  res.status(201).end();
});

app.delete('/words', getAccessToken, requireAccessToken, function(req, res) {
  savedWords.pop();
  res.status(204).end();
});
```

We're going to modify each of these to make sure that the scopes of the token contain at least the scope associated with each function. Because of the way our tokens are

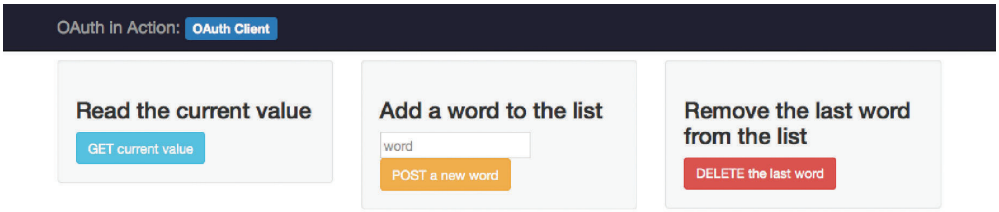


**Figure 4.4 The client with three different functions, each mapped to a scope**

stored in our data store, we have to get the `scope` member associated with the token. For the `GET` function, we want the client to have the `read` scope associated with it. It could have other scopes as well, but our API doesn't particularly care if it does or not.

```
app.get('/words', getAccessToken, requireAccessToken, function(req, res) {
  if (__.contains(req.access_token.scope, 'read')) {
       res.json({words: savedWords.join(' '), timestamp: Date.now()});
  } else {
       res.set('WWW-Authenticate', 'Bearer realm=localhost:9002,
  error="insufficient_scope", scope="read"');
       res.status(403);
  }
});
```

We send the error back in the `WWW-Authenticate` header. This tells the client that this resource will take an OAuth bearer token, and that it at least needs to have the scope `read` associated with it for the call to succeed. We'll add similar code to the other two functions, testing for the `write` and `delete` scopes, respectively. In all cases, if the token doesn't have the right scope, an error is returned even if the token is otherwise valid.

```
app.post('/words', getAccessToken, requireAccessToken, function(req, res) {
  if (__.contains(req.access_token.scope, 'write')) {
       if (req.body.word) {
              savedWords.push(req.body.word);
       }
       res.status(201).end();
  } else {
       res.set('WWW-Authenticate', 'Bearer realm=localhost:9002,
  error="insufficient_scope", scope="write"');
       res.status(403);
  }
});

app.delete('/words', getAccessToken, requireAccessToken, function(req, res) {
  if (__.contains(req.access_token.scope, 'delete')) {
       savedWords.pop();
       res.status(204).end();
  } else {
       res.set('WWW-Authenticate', 'Bearer realm=localhost:9002,
  error="insufficient_scope", scope="delete"');
       res.status(403);
  }
});
```

With that in place, reauthorize the client application to allow for different combinations of scopes. For example, try giving the client `read` and `write` access without `delete` access. You'll see that you can push data into the collection but you can never remove it. For an advanced take on this exercise, extend the protected resource and client to allow for more scopes and more types of access. Don't forget to update the client's registration in the authorization server for this exercise!

### 4.3.2 Different scopes for different data results

In this style of API design, different kinds of information can be returned from the same handler depending on which scopes are present in the incoming token. This is useful when you have a complex set of structured information and you want to be able to give access to subsets of the information without the client needing to call different API endpoints for each type of information.

Open up `ch-4-ex-3` and edit `protectedResource.js`, leaving `client.js` and `authorizationServer.js` alone. The client has a page that will allow you to call the API, once you get a token, and will display the list of produce that results (figure 4.5).

In the protected resource code, instead of multiple separate handlers (one for each type of produce), we have a single handler for all produce calls. At the moment, this returns an object containing a list of produce in each category.

```
app.get('/produce', getAccessToken, requireAccessToken, function(req, res) {
   var produce = {fruit: ['apple', 'banana', 'kiwi'],
        veggies: ['lettuce', 'onion', 'potato'],
        meats: ['bacon', 'steak', 'chicken breast']};
   res.json(produce);
});
```

Before we do anything else, if we were to fetch this API with any valid access token right now, we would always get back a list of all produce. If you authorize the client to get an access token, but don't allow it any scopes, you'll get back a screen that looks something like the one in figure 4.6.

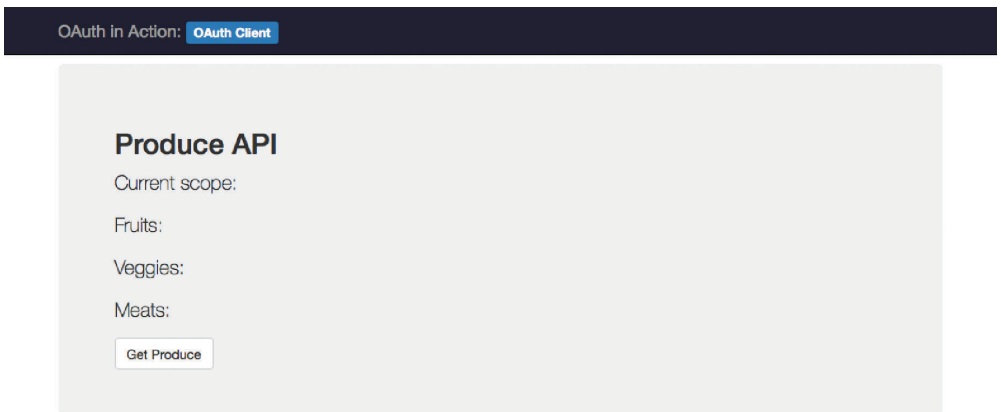


**Figure 4.5 The client's page before any data is fetched**

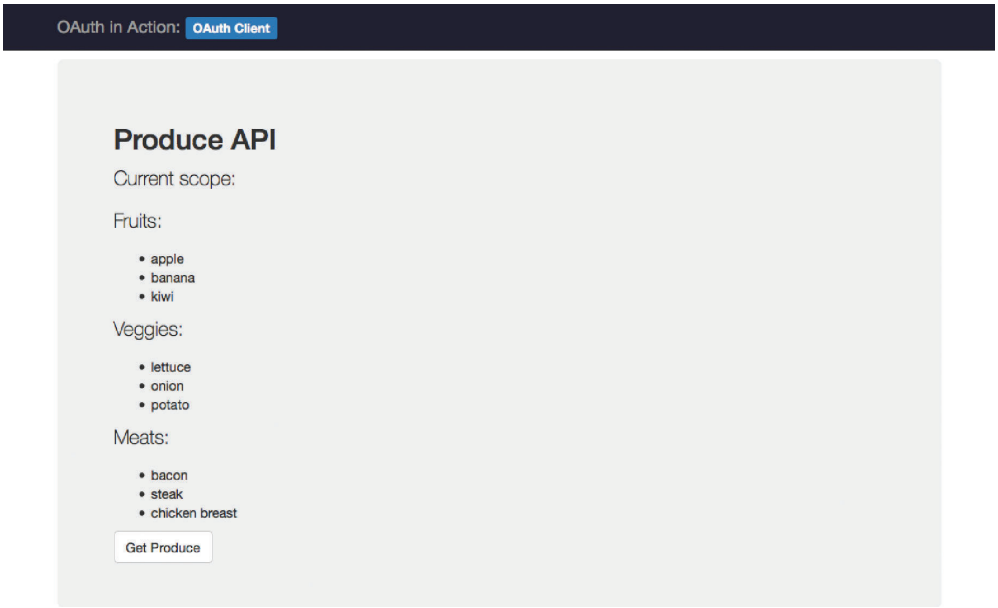


**Figure 4.6 The client's page showing all data returned with no scopes specified**

However, we want our protected resource to be able to split up the produce section based on scopes that were authorized for the client. First, we'll need to slice up our data object into sections so that we can work with it more easily.

```
var produce = {fruit: [], veggies: [], meats: []};
produce.fruit = ['apple', 'banana', 'kiwi'];
produce.veggies = ['lettuce', 'onion', 'potato'];
produce.meats = ['bacon', 'steak', 'chicken breast'];
```

Now we can wrap each of these sections into control statements that check for specific scopes for each produce type.

```
var produce = {fruit: [], veggies: [], meats: []};
if (__.contains(req.access_token.scope, 'fruit')) {
     produce.fruit = ['apple', 'banana', 'kiwi'];
}
if (__.contains(req.access_token.scope, 'veggies')) {
     produce.veggies = ['lettuce', 'onion', 'potato'];
}
if (__.contains(req.access_token.scope, 'meats')) {
     produce.meats = ['bacon', 'steak', 'chicken breast'];
}
```

Now authorize the client application for only the `fruit` and `veggies` scopes, and try the request again. You should get back a vegetarian shopping list (figure 4.7).[2]

[2] This removes all of the meats, even though, as we all know, bacon is sometimes a vegetable.

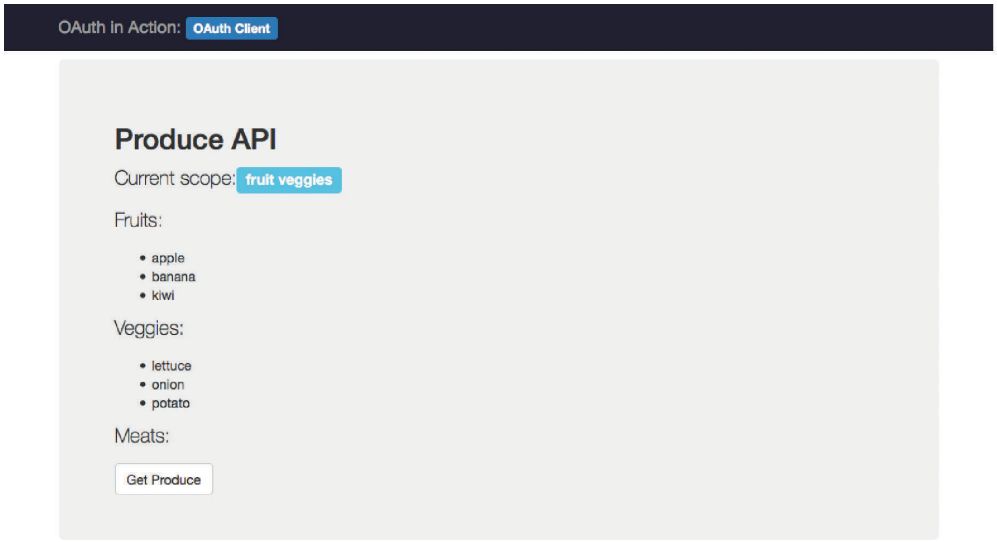


**Figure 4.7 The client's page showing limited data returned based on scopes**

Of course, nothing in OAuth requires us to split our API at high-level objects in this way. For an added exercise, add a `lowcarb` scope option to the client and resource server, returning only foods that are low in carbohydrates from each category. This can be combined with the type category scopes from the exercise above, or it can act on its own. Ultimately, it's up to you, the API designer, what the scope means. OAuth merely provides a mechanism for carrying it.

### 4.3.3 Different users for different data results

In this style of API design, different information is returned from the same handler depending on who authorized the client. This is a common approach to API design as it allows a client application to call a single URL with no knowledge of who the user is yet still receive individualized results. This is the kind of API used in our cloud-printing example from chapters 1 and 2: the printing service calls the same photo-storage API regardless of who the user is, and gets that user's photos. The printing service never needs to know a user identifier, or anything else about who the user is.

Open up `ch-4-ex-4` and edit `protectedResource.js`, leaving `client.js` and `authorizationServer.js` alone. This exercise will provide a single resource URL that returns information about a user's favorites in several categories based on who authorized the access token. Even though the resource owner isn't present or authenticated on the connection between the client and the protected resource, the token generated will contain a reference to the resource owner who was authenticated during the approval process (figure 4.8).

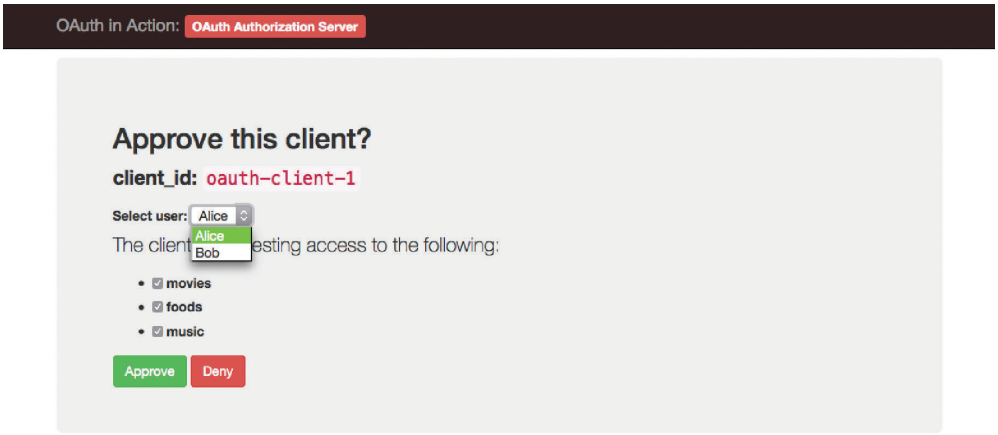


**Figure 4.8 The authorization server's approval page, showing a selection of resource owner identity**

### A drop-down menu is not authentication

The authorization server's approval page will let you select which user you're responding for: Alice or Bob. Normally, this would be handled by authenticating the resource owner at the authorization server, and it's generally considered especially bad security practice to allow an unauthenticated user to impersonate anyone of their choosing on the system. For testing purposes, though, we're keeping the example code simple and allowing you to choose the current user from a drop-down menu. For an additional exercise, try adding a user authentication component to the authorization server. Many different modules are available for Node.js and Express.js with which you can experiment.

The client has a page that will allow you to call the API, once you get a token, and will display the personalized information that results (figure 4.9).

At the moment, as you can see, it doesn't know which user you're asking about, so it's returning an unknown user with no favorites. Looking in the code of the protected resource, it's easy to see how that happened.

```
app.get('/favorites', getAccessToken, requireAccessToken, function(req, res) {
   var unknown = {user: 'Unknown', favorites: {movies: [], foods: [], music:
   []}};
   console.log('Returning', unknown);
   res.json(unknown);
});
```

It turns out that we have some information about Alice and Bob at the protected resource, and it's stored in the `aliceFavorites` and `bobFavorites` variables, respectively.

```
var aliceFavorites = {
   'movies': ['The Multidmensional Vector', 'Space Fights', 'Jewelry Boss'],
   'foods': ['bacon', 'pizza', 'bacon pizza'],
```

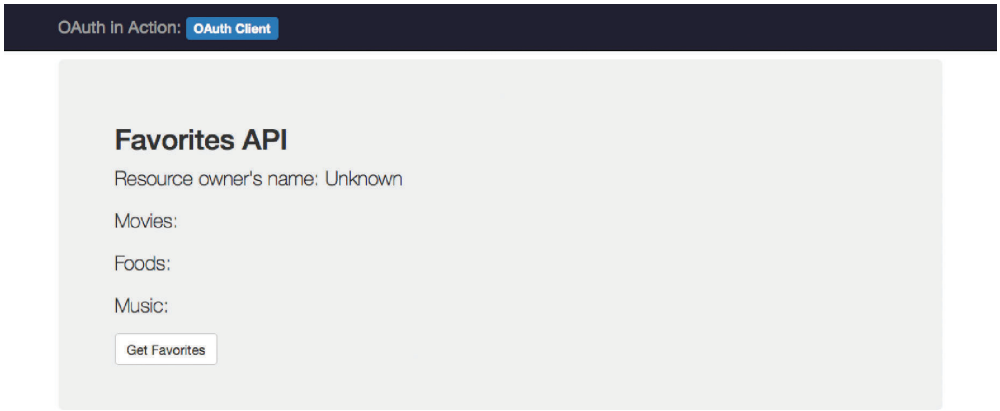


**Figure 4.9 The client's page before data has been fetched**

```
   'music': ['techno', 'industrial', 'alternative']
};

var bobFavorites = {
   'movies': ['An Unrequited Love', 'Several Shades of Turquoise', 'Think Of
   The Children'],
   'foods': ['bacon', 'kale', 'gravel'],
   'music': ['baroque', 'ukulele', 'baroque ukulele']
};
```

All we need to do, then, is dispatch which data record we want to send out based on who authorized the client. Our authorization server has stored the username of the resource owner in the user field of the access token's record, so all we need to do is switch out the content based on that.

```
app.get('/favorites', getAccessToken, requireAccessToken, function(req, res) {
   if (req.access_token.user == 'alice') {
        res.json({user: 'Alice', favorites: aliceFavorites});
   } else if (req.access_token.user == 'bob') {
        res.json({user: 'Bob', favorites: bobFavorites});
   } else {
        var unknown = {user: 'Unknown', favorites: {movies: [], foods: [],
   music: []}};
        res.json(unknown);
   }
});
```

Now if you authorize the client for either Alice or Bob at the authorization server, you should get their personalized data at the client. For example, Alice's list is shown in figure 4.10.

During the OAuth process, the client never knew that it was talking to Alice and not Bob or Eve or someone else entirely. The client only learned Alice's name incidentally because the API it was calling included her name in the response, and it could have just as easily been left out. This is a powerful design pattern, as it can protect the

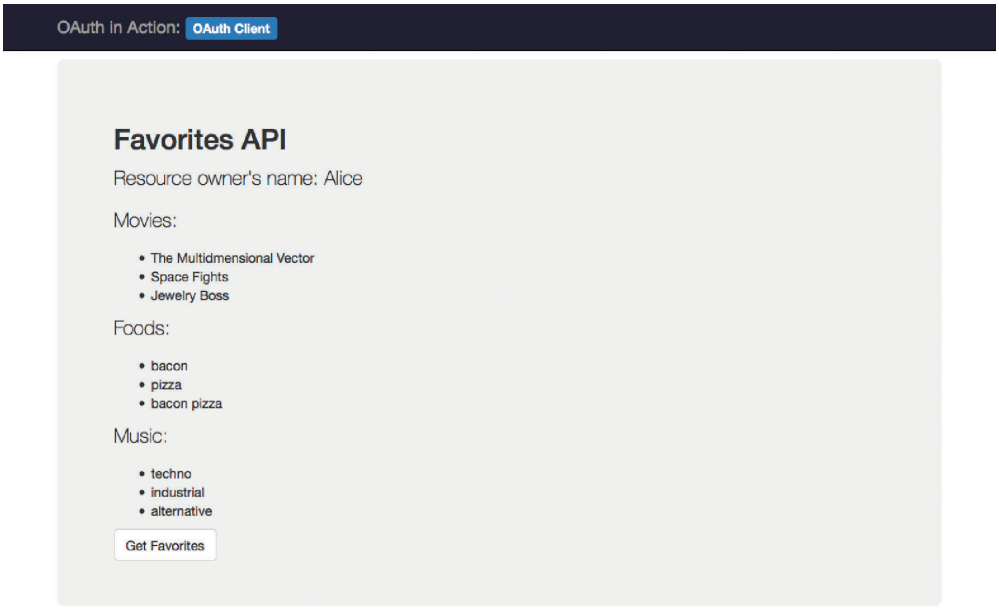


**Figure 4.10 The client's page showing Alice's resource data**

resource owner's privacy by not revealing personally identifying information unnecessarily. When coupled with an API that divulges user information, OAuth can start to approach an authentication protocol. We'll take a deeper look at that in chapter 13, including looking at the additional functions and features needed to support end-user authentication.

Of course, you can combine these methods. Our authorization server and client are already set up to use different scopes for this exercise, but the protected resource currently ignores them. As an added exercise, filter out the response to the favorites API based on the `movies`, `foods`, and `music` scopes for which the client has been authorized.

### 4.3.4 Additional access controls

This chapter's list of potential access controls that can be applied by a protected resource using OAuth is far from comprehensive, and there are perhaps as many specific patterns in use today as there are resources that are being protected. Because of this, OAuth itself has stayed out of the authorization decision-making process and instead acts as a carrier for authorization information through the use of tokens and scopes. This approach allows OAuth to be applied to a vastly diverse set of API styles across the internet.

A resource server can use the tokens, and the information attached to them such as scopes, and make authorization decisions directly based on that. Alternatively, a resource server can combine the rights associated with an access token with other

pieces of access control information in order to make a decision about whether to serve an API call and what to return for a given request. For example, a resource server can decide to limit certain clients and users to access things only during certain times, no matter whether the token is good or not. A resource server could even make a call to an external policy engine with the token as input, allowing complex authorization rules to be centralized within an organization.

In all cases, resource servers have the final say about what an access token means. No matter how much of their decision-making process they outsource, it's always up to the resource server to decide what to do in the context of any given request.

## 4.4 Summary

Protecting a web API with OAuth is fairly straightforward.

- The token is parsed out of the incoming request.
- The token is validated with the authorization server.
- The response is served based on what the token is good for, which can take several forms.

Now that you've built both the client and protected resource, it's time to build the most complex, and arguably most important, component of an OAuth system: the authorization server.